

OpenMP benchmark using PARKBENCH

Mitsuhsa Sato, Kazuhiro Kusano and Sigehisa Satoh
Real World Computing Partnership, Japan
E-mail: {msato, kusano, sh-sato}@trc.rwcp.or.jp

1 Introduction

In this paper, we propose an OpenMP benchmark using the PARKBENCH benchmark, and present some results and experience on the benchmark.

To understand the performance of OpenMP programs, the following factors are important:

1. The overheads of OpenMP constructs such as creating a team of threads, loop scheduling and synchronization.
2. OpenMP is an API for shared memory programming. While these overheads of OpenMP constructs are an important factor, the performance of parallel execution on the target shared memory platform is significant in determining the performance of OpenMP parallel programs.

Real application codes in OpenMP obviously measure the performance of OpenMP programming on the real problems. Although this is ultimately what the end-user wants, the full real applications are often complex and large. In order to obtain a guide to the performance of OpenMP parallel programs in any given parallel systems, kernel and synthetic benchmarks are useful.

PARKBENCH[4] is a set of benchmark programs proposed by PARKBENCH committee. It consists of synthetic low-level benchmarks, kernel benchmarks and compact applications. This hierarchical structure allows information derived from the simpler code to be used in explaining the performance characteristics of the more complicated codes. Currently, the low-level and kernel benchmarks are available. The kernel benchmarks includes some kernel programs from the NAS parallel benchmarks. In PARKBENCH, fundamental metrics such as the accuracy of clock and the flop count is clearly defined. In the current version, the parallel benchmarks are written using MPI and PVM to evaluate parallel program in distributed memory environments.

In this paper, we focus on the single processor benchmarks in the low-level benchmark of PARKBENCH as an OpenMP benchmarks. We have parallelized these benchmarks using OpenMP. To understand the performance of OpenMP programs, it is important to measure the performance of shared memory hardware platforms as well as the OpenMP

constructs overheads. The benchmark programs in OpenMP shows the performance of a set of loops parallelized by using OpenMP programming model. The benchmark assesses the significance of the overheads of OpenMP loop constructs in several kinds of loops. The overheads of the parallel constructs should be evaluated with respect to shared memory performance. This benchmark results show a guide to parallelize loops using OpenMP. For example, the OpenMP programmer can identify the effective loop-length in OpenMP for several kinds of loops.

Some OpenMP benchmarks were already proposed. EPCC OpenMP microbenchmark[2] is a synthetic benchmark to measure the overheads of synchronization primitives and loop scheduling for language constructs in OpenMP. Although this benchmark gives a valuable information about the overheads of OpenMP constructs, it does not address the performance of the shared memory hardware.

The NAS parallel benchmark suite[3] is the most popular benchmark for distributed memory platforms. Recently, a OpenMP version of the benchmarks were already prepared, and is expected to be available soon.

As realistic application benchmarks, One of SPECchpc96, SPECclimate is parallelized by OpenMP. SPEC HPG [6] has announced a plan of SPEC OpenMP benchmarks as one of SPEC benchmark suites for high performance computing field.

In the next section, we introduce the PARKBENCH benchmarks briefly and the OpenMP version of its low-level benchmark. Section 3 presents the results of the measurements on the different systems and different compilers, and the analysis by PARKBENCH outputs. We conclude our work in Section 4.

2 PARKBENCH for OpenMP benchmark

We use the low-level benchmarks of PARKBENCH as a OpenMP benchmark. The low-level benchmarks consist of the following programs:

- TICK1 and TICK2, which measure the timer resolution and value.
- RINF1, which measures the performance of basic arithmetic operations.

- PLOY1 and POLY2, which measure cache and memory bottleneck.
- COMM1, COMM2 and COMM3 which measure the performance of message passing in the distributed memory environments.
- PLOY3, a distributed memory version of POLY1.
- SYNC1, which measures barrier synchronization time of message passing.

RINF1, PLOY1 and PLOY2 are for a single processor, and COMM1, COMM2, COMM3, PLOY3 and SYNC1 are for a distributed memory multi-processors. RINF1, POLY1 and POLY2 are also in GENESIS benchmarks[1].

As an OpenMP benchmark, we focus on the single processor benchmarks. RINF1 characterizes the performance of basic arithmetic operations in several kinds of parallel loops. This benchmark takes a set of commonly used DO-loops and analyzes the time of execution in terms of the two parameter r_∞ (the asymptotic performance rate in MFlops/s) and $n_{1/2}$ (the half-performance length). Suppose that a loop (or vector) length is n and the loop contains q floating-point operations per element per iterations, the execution time T_n is approximated by:

$$T_n = (q \times n)/r_\infty + T_0$$

where T_0 is the loop startup overhead. Then the performance rate r_n at the loop length n is given by:

$$r_n = \frac{q \times n}{T_n} = \frac{r_\infty}{1 + (T_0 \times r_\infty / q \times n)}$$

$n_{1/2}$ is defined as the loop length to required to achieve 50 % performance of the the asymptotic performance rate r_∞ . The overhead T_0 is given by:

$$T_0 = \frac{n_{1/2} \times q}{r_\infty}$$

The parameter $n_{1/2}$ is a way to measuring the importance of vector startup overheads in the terms of quantities known to the programmer (loop or vector length). In a loop in OpenMP, the overhead T_0 includes the overhead of loop scheduling. In the benchmark program, the two parameter are determined by a least square fit of the data to the straight line defined by:

$$r_n = \frac{r_\infty}{1 + n_{1/2}/n}$$

The loops are parallelized by OpenMP for the measurement as follows:

```

T1 = DWALLTIME00()
!$OMP parallel private(JT)
DO JT=1,NTIM
CALL DUMMY(JT)
!$OMP do
DO I=1,N
... vector computation ...
END DO
!$OMP end do
!$OMP end parallel
T2 = DWALLTIME00()

```

The function DWALLTIME00 returns the wall clock time. The execution time is computed by $T = T2 - T1 - T0$. $T0$ is computed by a dummy loop in advance. Note that calling DUMMY is inserted to prevent optimizations.

The loops in RINF1 are listed in Table 2. The original loops are from 1 to 17. The loops from 18 to 20 are to measure the overhead of OpenMP loop scheduling methods, based on the loop 16 (DAXPY). The loop 18 uses PARALLEL DO instead of DO for the internal loop, and loop 19 and 20 specify static and dynamic scheduling respectively with chunk size 10.

POLY1 and POLY2 are to measure the basic hardware performance on shared memory access bottleneck. The computational intensity, f , of a DO-loop is defined as the number of floating point operations performed per memory reference to an element of a vector variable. The asymptotic performance, r_∞ , is observed to increase as the computational intensity increases, since the effect of memory access delayed become negligible compared the time spent on arithmetic. This is modeled by the two parameter, \hat{r}_∞ (the peek hardware performance of the arithmetic unit) and $f_{1/2}$ (the half computational intensity). The asymptotic performance r_∞ is given by:

$$r_\infty = \frac{\hat{r}_\infty}{1 + f_{1/2}/f}$$

Like $n_{1/2}$, $f_{1/2}$ is the computational intensity to achieve a half of the peek arithmetic hardware performance. Note that if memory access and arithmetic is not overlapped, $f_{1/2}$ shows the ratio of arithmetic speed to the memory speed.

The POLY1 benchmark repeats the evaluation of polynomials by Horner's rule for vector length up to 10,000 which would normally fits into the cache. It is therefore an in-cache test between the registers and the cache memory. The POLY2 is an out-of-cache tests which repeats the polynomial evaluation upto 100,000. To measure $f_{1/2}$, the order of the polynomial is increased, and the measured performance is fitted to the above equation.

To measure \hat{r}_∞ and $f_{1/2}$ of OpenMP loops, we parallelized these benchmarks as for RINF1.

No.	Arithmetic Ops.	Fops	Note
1	DYADS, $A(I)=B(I)*C(I)$	1.0	contiguous
2	DYADS, $A(I)=B(I)*C(I)$	1.0	stride=8
3	TRIADS, $A(I)=B(I)*C(I)+D(I)$	2.0	contiguous
4	TRIADS, $A(I)=B(I)*C(I)+D(I)$	2.0	stride=8
5	Random Scatter/Gather	2.0	
6	$A(I)=B(I)*C(I)+D(I)*E(I)+F(I)$	4.0	contiguous
7	Inner Product, $S=S+B(I)*C(I)$	2.0	single, reduction
8	First Order Recurrence	2.0	not parallelized
9	Charge Assignment: $A(J(I))=A(J(I))+S$	1.0	atomic
10	Transposition: $B(I,J)=A(J,I)$	N	
11	Matrix Mult BY Inner Product	$2*N*N$	
12	Matrix Mult BY Middle Product	$2*N*N$	
13	Matrix Mult BY Outer Product	$2*N$	
14	DYADS, $A(I)=B(I)*C(I)$	1.0	stride=128
15	DYADS, $A(I)=B(I)*C(I)$	1.0	stride=1024
16	DAXPY, $A(I)=S*B(I)+C(I)$	2.0	contiguous
17	DAXPY, $A(J(I))=S*B(K(I))+C(L(I))$	2.0	Indirect
18	DAXPY, $A(I)=S*B(I)+C(I)$	2.0	contiguous, parallel do
19	DAXPY, $A(I)=S*B(I)+C(I)$	2.0	contiguous, schedule(static,10)
20	DAXPY, $A(I)=S*B(I)+C(I)$	2.0	contiguous, schedule(dynamic,10)

Table 1: Description of kernel loops in RINF1 benchmark

3 Results and Analysis

We have run the OpenMP version of RINF1, POLY1 and POLY2 on the following systems and compilers:

- COMPAQ ProLiant 6500 (Intel Pentium II Xeon 450MHz, 1MB cache/CPU, 4CPU, 1GB main memory, Linux Red-Hat 6.0) using RWCP Omni Fortran77 OpenMP compiler[5] (a translator to C. the backend C compiler is egcs-2.91.66 with options -O3 -malign-double). This platform is indicated as 'Omni-Linux'.
- COMPAQ ProLiant 6500 using PGI Fortran OpenMP compiler (pgcc 3.1-2). This platform is indicated as 'PGI-Linux'.
- Sun Enterprise 450 (Ultra Sparc 300MHz, 4CPU, 1GB, Solaris 2.6) using RWCP Omni Fortran77 compiler (the backend C compiler is SUN Wspro 4.2 with options -fast). This platform is indicated as 'Omni-Sun'.

We have run the benchmarks varying the number of processors from one to four.

The RINF1 benchmark outputs several information including the execution time data for each kernel and loop length. Figure 1 shows the performance rate r_n of the kernel loop 6 on each platform with the number of processor 1, 2 and 4. Horizontal and vertical axis indicate the loop-length and the performance rate in Flops respectively for the number of processors 1,2,4.

As found in these figures, there are two asymptotic performance rates for in-cache case and out-cache case. When the vector length exceeds out of the cache size, the performance rates drops down due to cache miss. The least-square fitting routine in RINF1 detects this change of performance rates by checking the execution time, and restart the least square fitting from this point. The RINF1 output the summary of selected values in the measurement including the two two asymptotic performance rates for in-cache and out-of-cache, and the max and min values.

Table 4 shows the summary of RINF1 results on Omni-Linux. In Tables 4 and 4, a part of the summary on PGI-Linux and Omni-Sun is included. Comparing to the result on Omni-Linux, PGI-Linux achieves fairly better performance especially when the vector length is fitting in the cache size. Note that because the Omni compiler is a translator to C codes, the result of Omni reflects the performance of the backend C compiler. In the Sun platform, the differences between in-cache rate and out-of-cache rate is smaller than that in the Intel's platform.

As described in the previous section, the overheads of OpenMP constructs are included in $n_{1/2}$. If the performance rate increases with more processors, the $n_{1/2}$ values becomes larger and more large startup time is required to get performance effectively. In case of out-of-cache, the $n_{1/2}$ value becomes negative in many kernel loops. It means that the loop overhead becomes negligible when the vector length is large enough to incur cache-miss.

Figure 2 compares the different versions of DAXPY (Loop 16, 18, 19, 20) with four processors on each platform. The loop 18 ('PARALLEL DO') startups slower than the loop 16('DO'). This is also indicated as a larger $n_{1/2}$ value of cache-in case in the summary table. As the vector length becomes longer, the differences between different loop scheduling is getting smaller because that loop scheduling overhead can be negligible.

Table 4 shows the results of shared memory bottleneck analysis by POLY1 and POLY2. The POLY1 in-cache test shows that in th modern microprocessors cache-memory bandwidth is enough for arithmetic operation speeds of CPU. However, POLY2 out-of-cache test indicates especially in PGI-Linux that the Intel processor's shared memory system needs a higher computational intensity against to use its high performance CPU effectively.

4 Conclusion

In this paper, we have proposed an OpenMP benchmark using the low-level single processor benchmarks of PARKBENCH benchmark, and results and analysis are shown using this benchmark. The loop overheads including that of OpenMP loop scheduling are shown as loop or vector length in the terms of quantities known to the programmer. The cache effect can be clearly analyzed by the asymptotic performance rate analysis. The performance analysis of shared memory provides a valuable guide to OpenMP programmers as well as OpenMP construct overheads analysis.

We have also made a C OpenMP version of PARKBENCH low-level benchmarks. We are planning to make these PARKBENCH OpenMP benchmarks available at Netlib.

References

- [1] C.A. Addison, V.S. Getov, A.J.G Hey, R.W. Hockney and I.C. Walton, "The GENESIS Distributed-memory Benchmarks", Computer Benchmarks, J.J. Dongara and W. Gentsch (eds), Advances in parallel Programming, Vol. 8., pp. 257-27.1
- [2] J. M. Bull, "Measuring Synchronization and Scheduling Overheads in OpenMP", Proc. of EWOMP '99, pp. 99-105, 1999.
- [3] D. Bailery, et.al, "The NAS Parallel Benchmarks", Int. J. on Supercomputer Application, 5(3):63-73,1991.
- [4] R.W. Hockey, and M. Berry, (eds), "Public International Benchmark for Parallel Computer", PARKBENCH Committee: Report-1.
- [5] RWCP Omni OpenMP compiler System, <http://pdplab.trc.rwcp.or.jp/Omni/>.
- [6] Standard Performance Evaluation Corporation, <http://www.spec.or.jp>.

No.	#P	vlen(1)	r_∞	$n_{1/2}$	vlen(2)	r_∞	$n_{1/2}$	r_n min	r_n max
1	seq	<= 400	52.95	4.8	>= 800	9.74	-4414.7	9.64	53.67
	2	<= 8000	53.02	35.5	>= 30000	15.41	-15515.9	5.96	58.41
	4	<= 8000	81.03	0.1	>= 30000	18.38	-21211.0	4.31	97.46
2	seq	<= 50	49.53	1.4	>= 90	3.83	-554.4	3.58	48.95
	2	<= 90	63.42	92.7	>= 400	6.93	-879.4	6.17	33.33
	4	<= 1000	35.75	18.2	>= 5000	15.12	-2910.6	4.81	61.32
3	seq	<= 80	80.56	0.8	>= 300	15.27	-2981.9	15.26	87.98
	2	<= 800	114.20	79.9	>= 3000	26.71	-6478.7	11.99	102.97
	4	<= 3000	162.49	97.1	>= 7000	41.11	-9602.1	8.78	180.82
4	seq	<= 40	81.25	2.1	>= 80	5.35	-491.8	5.33	77.42
	2	<= 80	116.20	83.5	>= 300	8.37	-967.8	7.59	59.34
	4	<= 900	62.70	17.6	>= 4000	17.59	-3300.3	9.69	94.25
5	seq	<= 600	46.56	1.1	>= 1000	15.87	-4791.7	15.17	46.79
	2	<= 8000	52.64	17.5	>= 30000	22.81	-17599.7	6.08	58.86
	4	<= 8000	91.91	117.5	>= 30000	52.35	-491.9	4.60	96.01
6	seq	<= 90	102.62	1.1	>= 400	19.78	-2588.6	20.20	101.83
	2	<= 800	149.79	37.0	>= 3000	31.33	-6960.7	23.46	151.86
	4	<= 1000	331.29	165.1	>= 5000	47.27	-8693.8	17.85	278.73
7	seq	<= 500	99.02	2.0	>= 900	24.27	-5711.7	22.80	98.62
	2	<= 8000	62.76	188.9	>= 30000	36.79	-10951.7	2.92	61.70
	4	<= 10000	116.27	759.5	>= 50000	69.94	-3324.1	1.44	109.04
9	seq	<= 800	33.72	0.7	>= 3000	26.43	-1015.7	26.33	33.79
	2	<= 80	2.00	3.3	>= 300	1.44	-615.5	1.33	1.97
	4	<= 80	1.01	1.5	>= 300	0.88	-27.9	0.84	1.00
10	seq				>= 50	12.85	-43.2	14.35	60.53
	2	<= 80	34.85	4.8				18.68	33.39
	4				>= 50	23.50	-42.6	13.88	61.38
11	seq	<= 20	75.84	1.9	>= 60	41.90	-29.7	44.86	69.67
	2	<= 80	33.02	0.8				29.74	32.80
	4	<= 80	66.51	2.8				49.28	65.32
12	seq	<= 80	70.85	0.3				50.23	72.25
	2	<= 90	33.10	0.8	>= 400	30.79	-3.1	30.32	32.87
	4	<= 90	66.32	2.5	>= 400	57.98	-4.8	49.54	65.52
13	seq				>= 40	10.73	-3096.8	10.70	23.39
	2	<= 400	35.80	62.0	>= 60	20.01	-3378.0	20.04	33.12
	4				>= 1	54.11	-699.8	21.24	64.64
14	seq				>= 30	7.52	-27.7	6.96	44.46
	2				>= 1	16.91	8.5	6.16	18.04
	4				>= 1	36.58	53.9	4.81	34.04
15	seq				>= 30	7.45	-7.1	7.95	10.10
	2				>= 1	18.20	22.8	5.67	14.70
	4				>= 1	33.56	58.5	4.92	20.46
16	seq	<= 300	111.00	2.3	>= 700	18.56	-4606.4	18.18	110.20
	2	<= 8000	96.38	15.0	>= 30000	32.94	-14964.2	12.10	111.29
	4	<= 8000	165.11	69.2	>= 30000	66.69	-3101.9	8.42	187.62
17	seq	<= 80	123.20	2.7	>= 300	10.21	-1504.8	10.30	119.05
	2	<= 300	88.68	60.3	>= 700	11.63	-3480.7	11.82	73.64
	4	<= 800	149.69	136.4	>= 3000	14.30	-6877.4	9.78	130.71
18	2	<= 8000	97.85	227.8	>= 30000	31.33	-17117.0	3.36	96.71
	4	<= 9000	175.79	685.2	>= 40000	34.91	-27387.0	2.20	164.45
19	2	<= 800	61.04	39.3	>= 3000	23.70	-5772.4	12.27	57.80
	4	<= 8000	96.59	53.2	>= 30000	46.85	-6425.8	10.25	98.71
20	2	<= 8000	18.35	52.7	>= 30000	12.74	-9432.3	2.56	18.29
	4	<= 20000	13.33	57.5	>= 60000	10.76	-9723.3	1.15	13.52

Table 2: RINF1 summary of results on Omni-Linux

No.	#P	vlen(1)	r_∞	$n_{1/2}$	vlen(2)	r_∞	$n_{1/2}$	r_n min	r_n max
1	1	<= 500	65.72	10.7	>= 900	9.63	-4899.4	9.61	66.13
	2	<= 800	134.66	131.1	>= 3000	20.83	-5841.2	9.33	119.04
	4	<= 2000	228.20	353.3	>= 6000	35.89	-4893.0	5.62	183.46
5	1	<= 700	58.62	7.9	>= 2000	15.64	-6135.0	15.00	58.58
	2	<= 1000	81.23	67.6	>= 5000	31.65	-6253.4	9.52	75.70
	4	<= 8000	102.88	76.4	>= 30000	58.93	-2.4	5.68	115.67
6	1	<= 100	164.45	7.8	>= 500	20.86	-3130.6	21.48	155.21
	2	<= 800	249.93	36.0	>= 3000	31.35	-7970.1	31.16	272.29
	4	<= 1000	596.44	246.7	>= 5000	60.75	-7593.7	22.33	465.62
7	1	<= 800	107.90	16.1	>= 3000	24.49	-7688.6	23.53	107.15
	2	<= 8000	139.09	96.6	>= 30000	63.03	-11193.3	6.46	163.40
	4	<= 9000	259.27	762.7	>= 40000	156.56	23904.3	2.91	240.68
9	1	<= 900	8.14	0.7	>= 4000	7.69	-198.6	7.28	8.13
	2	<= 80	1.54	-0.4	>= 300	1.28	-32.9	1.28	1.59
	4	<= 80	1.21	1.1	>= 300	0.87	-77.5	0.87	1.22
10	1				>= 50	11.90	-40.7	13.24	35.52
	2	<= 60	54.84	2.7	>= 100	17.44	-68.1	22.20	68.89
	4				>= 80	24.46	-66.0	16.76	115.46
11	1	<= 20	62.92	1.9	>= 60	40.04	-25.1	41.61	58.71
	2	<= 20	128.00	2.6	>= 60	84.09	-20.9	86.69	115.05
	4	<= 90	222.73	5.2	>= 400	180.09	-6.6	119.40	220.28
16	1	<= 500	125.20	10.6	>= 900	19.31	-5091.3	19.19	122.20
	2	<= 800	247.96	120.2	>= 3000	44.22	-5887.6	18.28	217.41
	4	<= 2000	412.96	311.1	>= 6000	81.97	-1996.7	11.24	341.93
18	2	<= 900	219.67	197.5	>= 4000	43.08	-6962.8	10.40	183.32
	4	<= 8000	207.95	67.6	>= 30000	80.45	-4456.4	4.35	275.49
19	2	<= 800	99.66	28.3	>= 3000	28.18	-7065.6	23.72	100.00
	4	<= 8000	145.94	19.3	>= 30000	44.20	-23393.995	13.04	168.04
20	2	<= 10000	21.52	93.470	>= 50000	13.51	-15460.935	4.06	21.39
	4	<= 8000	14.36	59.509	>= 30000	11.56	-5000.789	1.92	14.30

Table 3: RINF1 summary of results on PGI-Linux (partial)

No.	#P	vlen(1)	r_∞	$n_{1/2}$	vlen(2)	r_∞	$n_{1/2}$	r_n min	r_n max
1	1	<= 600	42.27	18.9	>= 1000	20.62	-2965.8	8.15	50.25
	2	<= 8000	58.25	12.8	>= 30000	41.63	1727.2	4.11	81.68
	4	<= 8000	112.21	129.3	>= 30000	54.95	5262.3	2.87	131.05
5	1	<= 800	57.50	63.5	>= 3000	24.67	-2159.4	7.80	54.16
	2	<= 8000	54.40	46.1	>= 30000	43.55	-5266.6	3.87	60.90
	4	<= 9000	96.48	226.0	>= 40000	75.78	-323.2	2.83	96.11
6	1	<= 200	85.47	20.1	>= 600	22.74	-3961.2	21.04	79.11
	2	<= 800	159.51	86.1	>= 3000	63.66	-2694.4	15.26	149.19
	4	<= 1000	316.30	241.7	>= 5000	99.08	-3675.8	11.23	265.27
7	1	<= 8000	147.14	118.6	>= 30000	181.59	20963.9	8.02	151.75
	2	<= 30000	224.98	571.9	>= 70000	407.31	152418.1	2.28	246.33
	4	<= 40000	244.55	357.3	>= 80000	767.84	183760.5	1.16	346.71
9	1	<= 80	4.14	2.9	>= 300	3.40	167.2	3.03	4.42
	2	<= 80	2.81	5.0	>= 300	1.87	-595.5	1.70	2.88
	4	<= 300	1.45	5.5	>= 700	0.85	-2170.5	0.81	1.45
10	1	<= 30	136.92	24.4	>= 70	8.52	-66.6	10.76	76.77
	2	<= 40	159.61	125.9	>= 80	14.44	-69.9	9.60	91.50
	4	<= 80	-174.89	-299.9	>= 300			7.08	77.34
11	1	<= 20	75.29	1.7	>= 60	18.90	-50.7	22.42	70.99
	2	<= 20	141.95	2.0	>= 60	37.38	-50.4	44.47	132.66
	4	<= 30	286.93	6.0	>= 70	72.50	-53.0	88.22	243.04
16	1	<= 500	63.58	19.8	>= 900	28.43	-2024.7	15.33	67.00
	2	<= 1000	129.44	139.5	>= 5000	61.02	-423.6	8.01	118.90
	4	<= 5000	160.46	66.2	>= 9000	102.29	3867.2	5.84	199.02
18	2	<= 20000	73.31	98.7	>= 60000	63.55	-457.6	2.54	83.25
	4	<= 10000	139.30	505.0	>= 50000	87.96	-8901.0	1.39	142.05
19	2	<= 8000	40.96	5.8	>= 30000	27.79	-9396.0	9.82	43.51
	4	<= 8000	82.47	76.5	>= 30000	59.90	-7262.9	7.50	84.22
20	2				>= 1	14.37	150.1	2.58	14.63
	4	<= 20000	12.95	144.3	>= 60000	11.32	-6518.9	1.03	12.90

Table 4: RINF1 summary of results on Omni-Sun (partial)

platform	#P	POLY1(in-cache)		POLY2(out-cache)	
		r_∞	$f_{1/2}$	r_∞	$f_{1/2}$
Omni-Linux	1	80.28	-0.15	75.86	1.24
	2	160.65	0.01	152.60	1.27
	4	314.13	-0.09	409.09	5.84
PGI-Linux	1	189.81	0.39	233.66	8.30
	2	380.14	0.38	472.96	8.63
	4	761.34	0.43	978.93	9.25
Omni-Sun	1	98.71	0.23	95.89	1.20
	2	197.12	0.20	197.78	2.50
	4	390.89	0.19	349.96	2.02

Table 5: POLY1 and POLY2 results

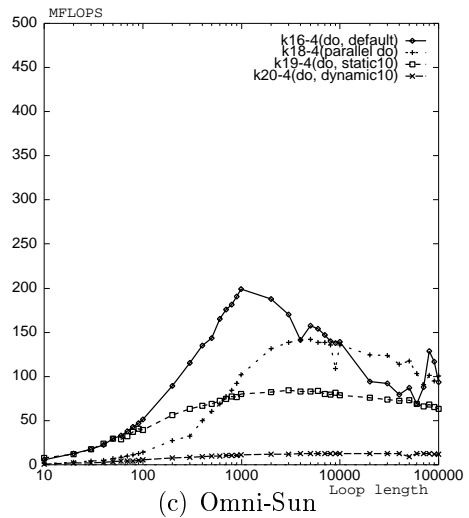
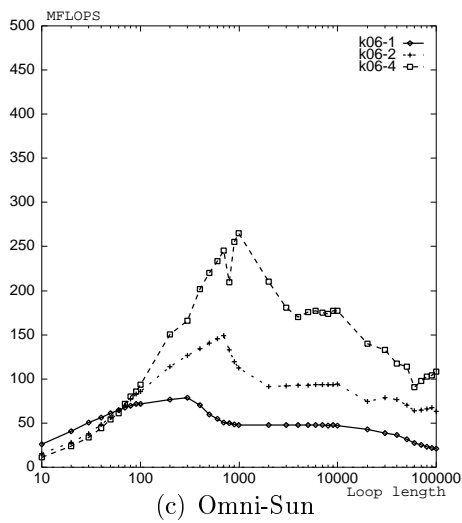
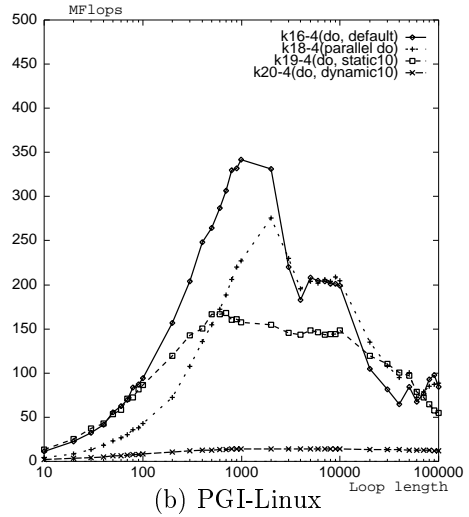
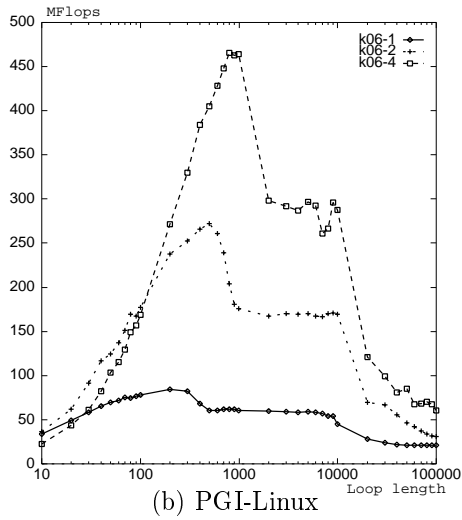
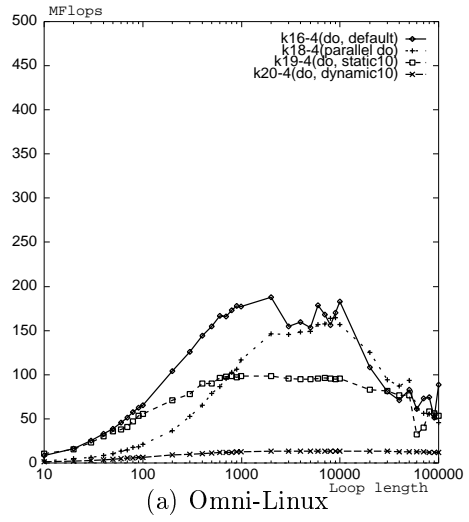
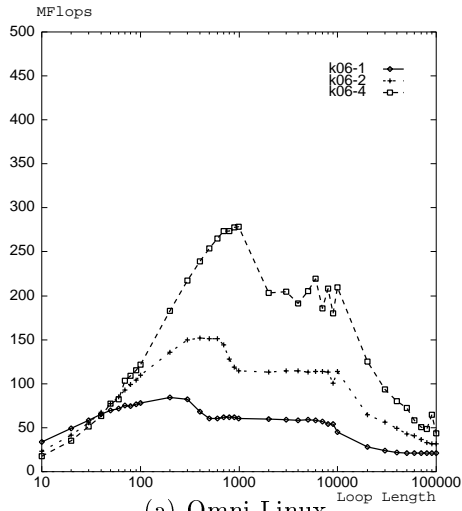


Figure 1: Performance rate r_n of Kernel Loop 6

Figure 2: Performance rate r_n of DAXPY with loop scheduling